Type-driven development in Cybersecurity and Business 4.0 with Idris

Christiano Braga

September, 2019

Universidade Federal Fluminense

COIN TCS // IC.UFF
TYPE-DIVEN CYBERSECURITY
WORKSHOPS

THE SECURITY YOU PRIMED FOR

Contact information

Christiano Braga
Associate Professor

Instituto de Computação
Universidade Federal Fluminense
cbraga@ic.uff.br
http://www.ic.uff.br/~cbraga
Lattes Curriculum Vitae

Introduction

Objective

- The objective of this workshop is to brainstorm about R&D opportunities between TCS and the Theoretical Computer Science Research Group at UFF, in particular exploring the type-driven development (TDD) approach.
- Our hypothesis is that the TDD approach can be **effectively** applied to either or both Cybersecurity and Business 4.0 enterprises at TCS with **clear ROI** as safety and security, for instance, would be increased in TCS solutions, based on public TCS documents.
  - TCS research website
  - Winning in a Business 4.0 World

Type-driven development in a nutshell

- Domain-specific languages
  - Focus on what is relevant to the client.
- Program transformation
  - Relates client terminology to the available solutions.
- Structural and behavioral type-safety
  - Allows for both *data* soundness and *process* soundness.
- Transparent use of rigorous program verification techniques.
  - Seamless integration of *mathematically rigorous* techniques into the development process.

Cybersecurity

- Current distributed applications ecosystem: IOT, Cloud, Web...
- A common problem in distributed information systems: *SQL code injection*.
  - Examples: Sony in 2011 and Yahoo! in 2012.
  - Losses of millions of dollars

The problem, by example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = "
        + txtUserId;
```

If `txtUserId` is equal to `105 OR 1=1`, which is always true, a malicious user may access *all* user information from a database.

Solutions

- SQL parameters: additional values are passed to the query.
- Escaping functions: they transform the input string into a "safe" one before sending it to the DBMS.
- The problem with the solutions is that communication relies on *strings*.
- What if we could **type** this information?

Protocols i

- Web programming invariably requires following certain **protocols**.
  - For example, to connect to make a query:
    1. Create a connection.
    2. Make sure the connection was established.
    3. Prepare an SQL statement.
    4. Execute the query.
    5. Process the result of the query.
    6. Close connection.
- Of course, a function could implement such a sequence, but how could one make sure that such a sequence is *always* followed?

Protocols ii

- In other words, what if we could *type* protocol behavior and make sure our Web programs *cope* with such types?
- Moreover, what if we could define special *notation* to create instances of such types?
- Protocols are one example but note that *business processes* may be treated the same way.

Service-oriented web development model i

Services are blackboxes, are stateless, are composable, among other nice characteristics.

- Services are first-class citizens in Cloud PaaS, and other platforms.
- These characteristics allow for a *clean* and *simple* interpretation of services as *functions*.
- ***What about capturing a company's way of developing PaaS as DSL?***
- ***What about capturing a company's clients processes as DSL?***

An example DSL

(From Fowler&Brady13.)

- Think of each step of a Web application as a business process.
- The notion of a Web application is typed, and so are its steps.
- For example, a Web application has forms and its forms have handlers.
- A particular Web application is *safe* (or well-typed) if its forms are well-typed. A form is well-typed if its handlers are also well-typed.

An example DSL ii

- The database protocol can be captured as a type.

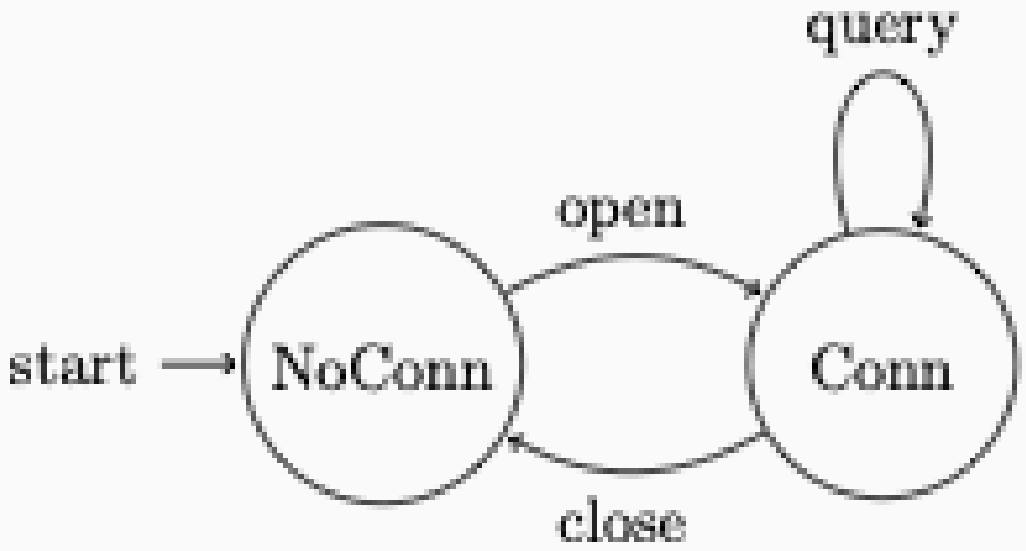

Figure 1: Database protocol

- A program that tries to make a query before opening a connection is **ill-typed**.
  - This is checked at compile time not run time!
  - Your client does not become aware of your errors!

Business 4.0

Critical business behaviors:

- Driving mass personalization
  - Personalizing products and services to a market of one customer, often even of one transaction, and at scale.
- Creating exponential value
  - Adopting business models that leverage value from transactions at multiple levels and address new markets.
- Leveraging ecosystems
  - Collaborating with partners inside and outside the supply chain to create new products and services.
- Embracing risk

  – Moving beyond rigid planning and operational barriers with an agile strategic approach.

Relating TDD and Buz4.0

- Mass personalization is domain-specific programming!
- Different business models may be captured as types and conformance to the business model becomes a programming practice!
- Type *composition* is natural in type-driven development!
- Safety and risk walk hand-in-hand as program transformation allows us to cope with agile strategies in a type-safe setting!

Our research approach

- To program with domain-specific languages, implemented on top of strongly typed functional languages.
- To develop and apply program analysis techniques to DSL-based approaches to software development.
- More specifically, to develop and apply cybersecurity and business 4.0 enabled-techniques in Idris.

This short-course

- In this short-course we will address some of the basic concepts of the type-driven approach that gives support to the development scenario outlined here.

Suggested reading

Edwin Brady. 2017. Type-driven development. Manning.

Simon Fowler and Edwin Brady. 2013. Dependent Types for Safe and Secure Web Programming. In Proceedings of the 25th symposium on Implementation and Application of Functional Languages (IFL '13). ACM, New York, NY, USA, Pages 49, 12 pages. DOI: https://doi.org/10.1145/2620678.2620683

The need for types

The need for types

- This section motivates the use of strong typing with a very very simple example: Bhaskara's theorem.
- In a tutorial way, we illustrate how types are necessary and, more specifically, how Idris' strong-typing presents itself as a powerful development tool.

Bhaskara's theorem

- From school: Bhaskara's theorem[1]

$$ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \stackrel{+}{-} \sqrt{\delta}}{2a}$$
$$\text{where } \delta = b^2 - 4acb$$

[1] For solving 2^{nd} degree polynomials. But this could might as well be an Excel formula, for instance! I mention Excel because that Microsoft is devoting serious efforts to develop a type system for Excel.

As functions

$$\begin{aligned}\texttt{bhask}(a, b, c) = & \\ & (-b + \sqrt{\texttt{delta}(a, b, c)}/2a, \\ & -b - \sqrt{\texttt{delta}(a, b, c)}/2a) \\ \texttt{delta}(a, b, c) = & \; b^2 - 4acb\end{aligned}$$

First attempt: no types i

- In Python:

```
from math import sqrt
def delta(a,b,c):
  return  (b * b) - (4 * a * c)
def bhask(a,b,c):
  d = delta(a,b,c)
  sr = sqrt(d)
  r1 = (-b + sr) / 2 * a
  r2 = (-b - sr) / 2 * a
  return (r1, r2)
```

First attempt: no types ii

- When we run `bhask(1,2,3)` the following is spit out:

```
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 9, in bhask
    sr = sqrt(d)
ValueError: math domain error
```

- This cryptic answer is only because we rushed into a direct implementation and forgot that `delta(a,b,c)` may return a *negative* value!

Second attempt: still no types. i

- Now, assuming we are interested only on Real results, how should `bhask` deal with the possibility of a negative `delta`?
- One possibility is to raise an *exception*:

```
from math import sqrt
def delta(a,b,c):
  return  (b * b) - (4 * a * c)
def bhask(a,b,c):
  d = delta(a,b,c)
  if d >= 0:
        sr = sqrt(d)
        r1 = (-b + sr) / 2 * a
        r2 = (-b - sr) / 2 * a
```

Second attempt: still no types. ii

```
        return (r1, r2)
    else:
        raise Exception("No Real results.")
```

- This implementation gives us a more *precise* answer:

```
Tue Jul 30@17:18:02:sc$ python3 -i bhask.py
Traceback (most recent call last):
  File "bhask.py", line 16, in <module>
    bhask(1,2,3)
  File "bhask.py", line 14, in bhask
    raise Exception("No Real results.")
Exception: No Real results.
```

Second attempt: still no types. iii

- A very **important** point here is that we only find all this out while actually *running* our implementation. Can't we do better? That is, let the **compiler** find out that `delta` may become a negative number and complain if this is not properly handled?

Third attempt: Idris. i

- Let us play with `delta` first.
- Strongly-typed languages, such as Idris, force us to think about types right away as we need to define `delta`'s signature. If we make the same mistake we did in the first attempt and forget that `delta` may become negative, we may write,

```
> delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Nat
> delta a b c = (b * b) - (4 * a * c)
```

the compiler would tell us:

Third attempt: Idris. ii

```
Type checking ./intro.lidr
intro.lidr:100:26:
    |
100 | > delta a b c =  (b * b) - (4 * a * c)
    |                          ^
When checking right hand side of delta with expected type
        Nat
When checking argument smaller to function Prelude.Nat.-:
     Can't find a value of type
     LTE (mult (plus a (plus a (plus a (plus a 0)))) c)
     (mult b b)
```

Third attempt: Idris. iii

- This is cryptic, in a first-glance, but tells us precisely **what** is wrong **and** at **compile** time. The problem is **with subtraction**: the type checker was not able to solve the inequality, defined in Idris' libraries,

$$4ac \leq b^2$$

in order to produce a **natural** number while computing `delta`, as natural numbers can not be negative!

First fix. i

- And we have not even started thinking about `bhask` yet! But let us first make `delta` type right by changing its signature:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (b * b) - (4 * a * c)
```

- To see the effect of this change, load `delta-fix.lidr` with the command:

```
:l delta-fix.lidr
```

- Don't be so happy though! This is not what we want yet.

First fix. ii

```
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
When checking right hand side of delta with expected type
        Int

Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
```

First fix. iii

- Idris does not know which subtraction operation to use because we are operating operating with natural numbers but we should return an integer! A casting is in order!

Second fix. i

- Think about why we should cast the right-hand side expression in the following way:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
```

and not the whole right-hand side of `delta` at once. - To see the effect of this change, load `delta-fix2.lidr` with the command:

```
:l delta-fix2.lidr
```

- You should finally be able to see

```
Type checking ./delta-fix2.lidr
*delta-fix2>
```

Second fix. ii

and run `delta 1 2 3`, for instance, to see the following result.

```
*delta-fix2> delta 1 2 3
-8 : Int
```

The road so far i

Your session should look like this at this point:

```
Mon Aug 05@14:24:16:the-need-for-types$
idris --nobanner tnft.lidr
Type checking ./tnft.lidr
tnft.lidr:107:25:
    |
107 | > delta a b c = (b * b) - (4 * a * c)
    |                         ^
When checking right hand side of delta with expected type
        Nat

When checking argument smaller to function Prelude.Nat.-:
```

The road so far ii

```
      Can't find a value of type
              LTE (mult (plus a (plus a (plus a (plus a 0))))
                  (mult b b)


Holes: Main.delta
*tnft> :l delta-fix.lidr
Type checking ./delta-fix.lidr
delta-fix.lidr:5:18-38:
  |
5 | > delta a b c =  (b * b) - (4 * a * c)
  |                  ~~~~~~~~~~~~~~~~~~~~~
When checking right hand side of delta with expected type
        Int
```

The road so far iii

```
Can't disambiguate since no name has a suitable type:
        Prelude.Interfaces.-, Prelude.Nat.-

Holes: Main.delta
*delta-fix> :l delta-fix2.lidr
*delta-fix2> delta 1 2 3
-8 : Int
```

Bhaskara at last! i

- Painful, no?
  No!
- The compiler is our *friend* and true friends do not always bring us good news!
- Think about it using this metaphor: do you prefer a shallow friend, such as Python, that says yes to (almost) everything we say (at compile time), but is not there for us when we really need it (at run time), or a *true* friend, such as Idris, that tells us that things are not all right all the time, but is there for us when we need it?

Bhaskara at last! ii

- Another way to put it is that "With great power comes great responsibility!", as the philosopher Ben Parker used to say... Strong typing, and in particular this form of strong typing, that relies on *automated theorem proving* requires some effort from our part in order to precisely tell the compiler how things should be.
- Having said that, let us finish this example by writing `bhask` function.

Bhaskara: first attempt i

- Bhaskara's solution for second-degree polynomials gives no Real solution (when $\delta < 0$), one (when $\delta = 0$), or two (when $\delta > 0$). Since "The Winter is Coming" we should be prepared for two roots:

```
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat)
                                -> (Double, Double)
bhask a b c = ((-b + (sqrt (delta a b c))) / (2 * a),
               (-b - (sqrt (delta a b c))) / (2 * a))
```

- Moreover, we should now work with the Idris `Double` type, because of the `sqrt` function. Run

Bhaskara: first attempt ii

```
*bhask-fun> :t sqrt
sqrt : Double -> Double
```

- Again, our naivete plays a trick on us:

```
Type checking ./bhask-fun.lidr
bhask-fun.lidr:2:19:
  |
2 | > bhask a b c =
            ((-b + (sqrt (delta a b c))) / (2 * a),
             (-b - (sqrt (delta a b c))) / (2 * a))
  |              ^
When checking right hand side of bhask with expected type
        (Double, Double)
```

Bhaskara: first attempt iii

```
When checking an application of function
    Prelude.Interfaces.negate:
        Type mismatch between
                Nat (Type of b)
        and
                Double (Expected type)
```

Load file `bhask-fun.lidr` to see this effect.

- We should write `negate b` instead of `- b`, as `-` is a *binary* operation only in Idris. Moreover, we should *not* be able to negate a natural number! Again, casting is necessary.

Bhaskara: final attempt i

- Let us fix all casting problem at once, the final definitions should be as follows:

```
delta : (a :  Nat) -> (b : Nat) -> (c : Nat) -> Int
delta a b c =  (cast (b * b)) - (cast (4 * a * c))
bhask : (a :  Nat) -> (b : Nat) -> (c : Nat)
                                -> (Double, Double)
bhask a b c =
  (negate (cast b) +
    (sqrt (cast (delta a b c))) / cast (2 * a),
   negate (cast b) -
    (sqrt (cast (delta a b c))) / cast (2 * a))
```

Bhaskara: final attempt ii

- We can now play with `bhask`, after executing `:l bhask-fun-fix.lidr`

```
Type checking ./bhask-fun-fix.lidr
*bhask-fun-fix> bhask 1 10 4
(-5.41742430504416, -14.582575694955839) :
                                (Double, Double)
*bhask-fun-fix> bhask 1 2 3
(NaN, NaN) : (Double, Double)
```

Bhaskara: final attempt iii

- Note that when $\delta < 0$ Idris gives a `NaN` value, which stands for *Not a number*. In other words, `bhask` is **total** as opposed to the **partial** approach in Python where we needed to raise an exception to capture the situation where the roots are not Real numbers.
- Idris can help us identify when a function is total. We simply need to run:

```
*bhask-fun-fix> :total bhask
Main.bhask is Total
```

Wrapping-up i

- First and foremost motivate strong-typing in Idris.
- Introduce notation for functions in Idris. The signature of a function, such as `delta` includes a name, formal parameters and a return type, such as:
  `delta : (a : Nat) -> (b : Nat) -> (c : Nat) -> Int`.
- The formal parameters of a function are declared using the so-called Currying form (after Haskell Curry): currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Wrapping-up ii

- This allows to *partially apply* a function! For instance, we can call `delta 1 2`. This will produce a function that expects a number and then behaves as `delta`.
- Take a look at the following session:

```
*bhask-fun-fix> delta
delta : Nat -> Nat -> Nat -> Int
*bhask-fun-fix> delta 1
delta 1 : Nat -> Nat -> Int
*bhask-fun-fix> delta 1 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> delta 1 2 3
-8 : Int
```

Wrapping-up iii

```
*bhask-fun-fix> (delta 1) 2
delta 1 2 : Nat -> Int
*bhask-fun-fix> ((delta 1) 2) 3
-8 : Int
```

- At the end of the day, `delta 1 2 3` is just *syntax sugar* for `((delta 1) 2) 3`.
- Total functions are such that, for all well-typed inputs, does one of the following:
  - Terminates with a well-typed result.
  - Produces a non-empty finite prefix of a well-typed infinite result in finite time. We can describe total functions as either terminating or productive.

Wrapping-up iv

  - The halting problem is the difficulty of determining whether a specific program terminates or not, and, thanks to Alan Turing, we know that it's impossible in general to write a program that solves the halting problem.
  - In other words, Idris can't determine whether one of these conditions holds for all total functions. Instead, it makes a conservative approximation by analyzing a function's syntax.

- Type casting. We have used `cast` many times in order to *inject* our values from one type into another.
- Some Read-Eval-Print-Loop (REPL) commands. We have seen how to load a file with `:l`, check its type with `:t`, and check weather a function is total or not with `:total`.

The need for dependent types

The need for dependent types

- Overflow conditions in software appear to be a simple thing to implement. An important counter-example is the Ariane 5 rocket that exploded due to a down cast from 64-bit number into a 16-bit one.
  The Ariane 5 had cost nearly $8 billion to develop, and was carrying a $500 million satellite payload when it exploded.
  11 of the most costly software errors in history
- In this chapter we look at a simplified version of the `Vector` datatype, available in Idris' library, to try and understand how *dependent typing* can be useful to have type-safe array handling that could help prevent catastrophes such as the Ariane 5 explosion.

Vector i

- A datatype is nothing but an implementation of some “domain of information”. It could very well represent low level information such as data acquired by a sensor in a Internet of Things (IoT) system or the structure that organizes the decision making process in planning.
- Our datatype here is quite simple but illustrates very well how dependent types may help safe data modeling and implementation.

Vector ii

```
> module Vect
> data Vect : Nat -> Type -> Type where
>     Nil  : Vect Z a
>     (::) : (x : a) -> (xs : Vect k a) -> Vect (S k) a
```

- An array or vector is built or *constructed* using either one of the constructor operations (unary) `Nil` or (binary) `::`. (The `module` keyword here simply defines a *namespace* where `Vect` will live.) After loading this file in Idris you could try

```
*tnfdt> 1 :: Vect.Nil
[1] : Vect 1 Integer
```

at the REPL.

`Vector` iii

- This says that the term `[1]` has type `Vect 1 Integer` meaning that it is a vector with one element and that its elements of the `Integer` type, Idris' basic types.
- Maybe this is a lot to take! *Just breath* and let us think about it for a moment.
- Types are defined in terms of constructor operators. This means that an *instance* of this type is written down as `1 :: Vect.Nil`. In a procedural language you could write it with a code similar to

```
v = insert(1, createVect(1))
```

where `createVect` returns a vector of a given size and `insert` puts an element on the given vector. The point is that we usually create objects or allocate memory to represent data in variables (so called *side effects*) while in functional programming we *symbolically* manipulate them, as in the example above.

- This is a major paradigm-shift for those not familiar with functional programming. Be certain that it will become easier as time goes by, but let's move on!

Dependency i

- Let's look at the instance first and then to the type declaration. Note that the type of `[1]` is `Vect 1 Integer`. The type of a Vect *depends* on its *size*! Think about examples of vectors in programming languages you know. If you query for the type of a given vector, if at all possible, what the run-time of your programming language will answer?
- In Python, for instance, you would get something like,

```
v = [1,2,3]
type(v)
<class 'list'>
```

Dependency ii

that is, is a `list` and that's all! In C an array is a pointer! (A reference to a memory address, for crying out loud!)

- In Idris, we know it is a vector and its size, an important property of this datatype. Cool! And so what?
- We can take advantage of that while programming. We could write a function that does *not*, under no circumstances, goes beyond the limits of a vector, that is, index it beyond its range!

The `zip` function i

- The `zip` function simple creates pairs of elements out of two instances of `Vect` *with the same size*. Here is what it look like:

```
> zip : Vect n a -> Vect n b -> Vect n (a, b)
> zip Nil Nil = Nil
> zip (x :: xs) (y :: ys) = (x, y) :: zip xs ys
```

- What on earth is it? Do you remember how to declare a function in Idris? Well, is pretty-much that. The difference here is that we are now programming with *pattern matching*.
- And what is it? Simply define a function by *cases*.

The `zip` function ii

- When we hit an instance of `Vect`, how does it look like? It is either the empty vector, built with constructor `Nil`, or a non-empty vector, built using operator `::`.
- These two cases are represented by each equation above. The first equation declares the case of "zipping" two *empty* vectors and the second one handles two *non-empty* vectors, specified by the *pattern* `x :: xs`, that is, a vector whose first element is `x` and its remaining elements are represented by a (sub)vector `xs`.
- For instance, if we could write

```
*tnfdt> Vect.zip [1,2,3] ["a", "b", "c"]
[(1, "a"), (2, "b"), (3, "c")] :
     Vect 3 (Integer, String)
```

The `zip` function iii

and get the expected vector of pairs produced by `zip`. (I used `Vect.zip` only because there are other `zip` functions coming from Idris' standard library.)

- Note that the type of `[(1, "a"), (2, "b"), (3, "c")]` is `Vect 3 (Integer, String)` where 3 is the size of the vector and `(Integer, String)`, denoting pairs of integers and strings, is the type of the elements of vector that `zip` calculates.

The `zip` function iv

- Note some additional interesting things about `zip`'s declaration: The signature of `zip` is `zip : Vect n a -> Vect n b -> Vect n (a, b)`. The variable `n` here stands for the size of the vector. Variables `a` and `b` denote the types of the elements of the vectors being zipped.
- That is, the `Vect` type is *generic*, as the type of its elements are underspecified, and is *dependent* on the **number** denoting its size. Again, `n` is a *number*, and `a` (or `b`, for that matter) is a *type*!
- Now, take a look at this:

The `zip` function v

```
*tnfdt> Vect.zip [1,2,3] ["a", "b"]
(input):1:19-21:When checking argument xs to
  constructor Vect.:::
        Type mismatch between
                Vect 0 a (Type of [])
        and
                Vect 1 String (Expected type)

        Specifically:
                Type mismatch between
                        0
                and
                        1
```

The `zip` function vi

- What does this mean? This is a *type checking* error, complaining about an attempt to zip vectors of different sizes. This is *not* an exception, raised while trying to execute `zip`. This is a *compile* type message, regarding the case of zip a vector of length 1 (the last element of the first vector), and a 0-sized vector (from the second vector).

In Idris, types can be manipulated just like any other language construct.

Conclusion.

Ariane 5 would not have exploded (from the bit conversion perspective) if the function that accidentally cast a 64-bit vector into a 16-bit one was written with this approach.

Wrapping-up

1. Defining datatypes.
2. Defining dependent datatypes.
3. Using dependent datatypes to find errors at compile time.
4. Type expressions.

Infinite data and processes

Infinite data i

- Streams are infinite sequences of values, and you can process one value at a time.
- When you write a function to generate a Stream, you give a prefix of the Stream and generate the remainder recursively. You can think of an interactive program as being a program that produces a potentially infinite sequence of interactive actions.

Infinite data ii

```
> %default total
> data InfIO : Type where
>     Do : IO a -> (a -> Inf InfIO) -> InfIO
> (>>=) : IO a -> (a -> Inf InfIO) -> InfIO
> (>>=) = Do
> loopPrint : String -> InfIO
> loopPrint msg = do putStrLn msg
>                    loopPrint msg
> partial
> run : InfIO -> IO ()
> run (Do action cont) = do res <- action
>                           run (cont res)
```

Infinite data iii

- Try the following at the REPL:

```
:exec run (loopPrint "on and on and on...")
```

and a non-terminating execution will present itself. As expected, `run` is *not* total:

```
*streams/streams> :total run
Main.run is possibly not total due to recursive path:
    Main.run, Main.run
```

- The type `InfIO`, as the name suggests, is a type of infinite IO actions, denoted by the type variable `a`. The `Do` constructor receives an IO action and produces an infinite IO action, by recursion.

Infinite data iv

- Function `loopPrint` is one such *action generator*.
- Let us take this slowly: First of all, what is the `Inf` type?

```
Inf : Type -> Type
Delay : (value : ty) -> Inf ty
Force : (computation : Inf ty) -> ty
```

- `Inf` is a generic type of potentially infinite computations.
- `Delay` is a function that states that its argument should only be evaluated when its result is forced.
- Force is a function that returns the result from a delayed computation.

Another example with infinite data i

- `InfList` is similar to the List generic type, with two significant differences:
  - There's no `Nil` constructor, only a `(::)` constructor, so there's no way to end the list.
  - The recursive argument is wrapped inside `Inf`.

```
> data InfList : Type -> Type where
>     (::) : (value : elem) -> Inf (InfList elem) ->
>     InfList elem
```

- Function `countFrom` is an example on how to use `Inf`.

Another example with infinite data ii

```
> countFrom : Integer -> InfList Integer
> countFrom x = x :: Delay (countFrom (x + 1))
```

The Delay means that the remainder of the list will only be calculated when explicitly requested using Force.

Try the following at the REPL:

```
*streams> countFrom 0
0 :: Delay (countFrom 1) : InfList Integer
```

Streams i

- Idris has streams in its prelude.

```
data Stream : Type -> Type where
   (::) : (value : elem) -> Inf (Stream elem) ->
                            Stream elem
repeat : elem -> Stream elem
take : (n : Nat) -> (xs : Stream elem) -> List elem
iterate : (f : elem -> elem) -> (x : elem) -> Stream elem
```

- Execute

Streams ii

```
 (iterate (+1) 0)
*streams/streams> (iterate (+1) 0)
0 ::
Delay (iterate (\ARG => prim__addBigInt ARG 1) 1) :
                                        Stream Integer
```

and try to grasp which type is this.

- Here are some cool stuff we can do with streams, try it out:

```
Idris> take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : List Integer
```

The syntax [1..] generates a Stream counting upwards from 1.

Streams iii

- This works for any countable numeric type, as in the following example:

```
Idris> the (List Int) take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : List Int
```

or

```
Idris> the (List Int) (take 10 [1,3..])
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19] : List Int
```

- Now, which is the relationship between all this machinery and the motivation presented at the beginning of the course?
  - Are there any relations among IOT sensors and streams?

Streams iv

- You should probably have realized by now that `run` is an *infinite process* executing on an *infinite stream* of data!

Making infinite processes total i

- As trivial as it may sound, a way to make a function terminate is simply to define a "time out".
- In the following example, this is denoted by the `Fuel` datatype. The `Lazy` datatype is similar to the `Inf` we have seen before, it "encapsulates" infinite data and only computes it when necessary.

```
> data Fuel =
>  Dry | More (Lazy Fuel)
>
> tank : Nat -> Fuel
```

Making infinite processes total ii

```
> tank Z = Dry
> tank (S k) = More (tank k)
>
> partial
> runPartial : InfIO -> IO ()
> runPartial (Do action f) =
>            do res <- action
>               runPartial (f res)
>
> run2 : Fuel -> InfIO -> IO ()
> run2 (More fuel) (Do c f) =
>      do res <- c
>         run2 fuel (f res)
```

Making infinite processes total iii

```
> run2 Dry p = putStrLn "Out of fuel"
>
> partial
> main : IO ()
> main = run2 (tank 10) (loopPrint "vroom")
```

`Inf` vs. `Lazy`

- If the argument has type Lazy ty, for some type ty, it's considered smaller than the constructor expression.
- If the argument has type Inf ty, for some type ty, it's not considered smaller than the constructor expression, because it may continue expanding indefi- nitely. Instead, Idris will check that the overall expression is productive

Protocols

A trivial database protocol

- The automaton below illustrates the communication between an application and a database system. The intention is to express that in order to query a database it is necessary first to establish a connection with it and then after all queries were done, the connection is closed.

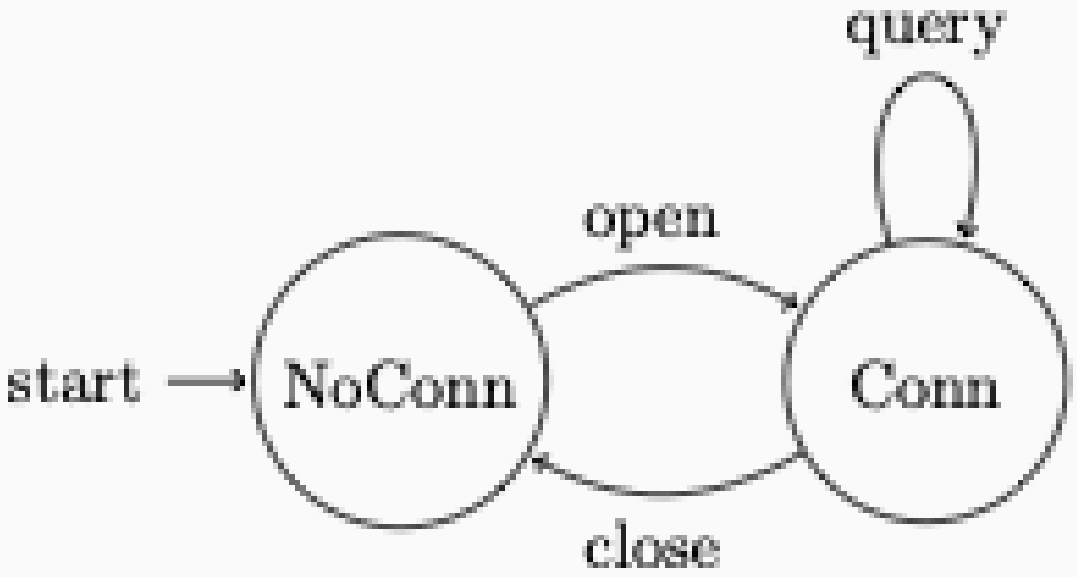

Figure 2: Trivial database protocol

First attempt: a monoid of actions i

- The code below is a naïve implementation of it.

```
> module DBProtocol
>
> import Data.Vect
>
>
> data DBConnState = Conn | NotConn
>
> namespace DBCmd1
>
>     data DBCmd : Type -> Type where
```

First attempt: a monoid of actions ii

```
>           Open : DBCmd ()
>           Close : DBCmd ()
>           Query : DBCmd ()

>           Pure : ty -> DBCmd ty
>           (>>=) : DBCmd a -> (a -> DBCmd b) -> DBCmd b
```

- Program `dbProg1` does exactly that.

```
>     dbProg1 : DBCmd ()
>     dbProg1 = do Open
>                  Query
>                  Close
```

First attempt: a monoid of actions iii

- But `dbProg2` also type checks just fine. Think about it for a moment. *Why is this the case?*

```
>     dbProg2 : DBCmd ()
>     dbProg2 = do Close
>                  Open
>                  Query
```

- Transitions are not *typed*! We can combine them in any way we want, in the code. But this is not the "spirit" of the specification. (Mathematically speaking, we do *not* want a *free* monoid of actions but rather an *ordered* one!)

Second attempt: a partial order i

- We can do better and we will. We can type transitions by annotating, each operation in `DBCmd` type, with the *source* and *target* types.
- This is captured in the type with signature

```
data DBCmd : Type -> DBConnState -> DBConnState -> Type.
```

- On each transition, for instance in `Open`, with the following signature:

```
Open : DBCmd () NotConn Conn
```

where `DBCmd ()` is its (returning) type.

Second attempt: a partial order ii

- Types `NotConn` and `Conn` are the types of the source and target states that specify, respectively, the pre and postconditions of the `Open` action.

```
> namespace DBCmd2
>
>     data DBCmd : Type -> DBConnState ->
>                          DBConnState -> Type where
>          Open : DBCmd () NotConn Conn
>          Close : DBCmd () Conn NotConn
>          Query : DBCmd () Conn Conn
>
>          Pure : ty -> DBCmd ty state state
```

Second attempt: a partial order iii

```
>           (>>=) : DBCmd a state1 state2 ->
>                   (a -> DBCmd b state2 state3) ->
>                   DBCmd b state1 state3
>
>     dbProg1 : DBCmd () NotConn NotConn
>     dbProg1 = do Open
>                  Query
>                  Close
```

- The sequence of actions `Open`, `Query` and `Close` types correctly, as expected.

Second attempt: a partial order iv

```
dbProg2 : DBCmd () NotConn NotConn
dbProg2 = do Query
             Close
             Open
```

- However, if a program tries to query a database to which there is no open connection, the program simply does not type-check!
- We can check it simply using command

```
idris --check protocol.lidr
```

Second attempt: a partial order v

as, in this example, there are not implementations for `Query`, `Close` and `Open`.

```
Tue Aug 13@16:06:57:protocols$
idris --check protocol.lidr
protocol.lidr:89:20-24:
   |
89 | >      dbProg2 = do Query
   |                     ~~~~~
When checking right hand side of
DBProtocol.DBCmd2.dbProg2
  with expected type
        DBCmd () NotConn NotConn
```

Second attempt: a partial order vi

```
When checking an application of constructor
DBProtocol.DBCmd2.>>=:
        Type mismatch between
                DBCmd () Conn Conn (Type of Query)
        and
                DBCmd a NotConn state2 (Expected type)

        Specifically:
                Type mismatch between
                        Conn
                and
                        NotConn
```

A simple app

Introduction i

- In this section we build on top of the implementation of the `Database` protocol we have just created.
- Before, we were interested, essentially, on specifying a datatype that captures the *behavior* (or automaton) of the protocol, guaranteeing that a computation (or transition) takes place only when its contract (pre and postconditions) hold.
  - In other words, we specify when *computations* are *well-formed*.
- Now we wish to write a *running* application on top of it. It has a command-line interface and uses a table or map (`SortedMap`, in Idris) to represent a database.

Putting it all together i

- Our app requires a few extensions with respect to what we have done so far:
  1. A way to transform string input into "commands" (a well-formed instance of a datatype.)
  2. A way to represent the database.
  3. A way to evaluate commands in the presence of a database.
  4. An updated protocol datatype that takes into account queries and reports.
  5. An interactive user-interface.
- You should note that we are essentially putting everything we studied together.

A way to transform string input into “commands” i

- Our app will simply open a database, close a database and query it. So let us first define a datatype that captures these three commands, and call it `Input`.

```
data Input = OPEN String
           | CLOSE
           | QUERY QueryLang
```

- A query should not be defined as a string, as always, we should type it! Of course, we will not define SQL here but focus on three commands:
  - `INSERT` adds an entry to the database composed by an `Integer` and a `String`.

A way to transform string input into “commands” ii

- `SELECT` retrieves the String bound to the given integer.
- `DELETE` removes from the database the entry whose key is the given integer.

- The `QueryLang` datatype implements it.

```
data QueryLang = INSERT Int String
               | SELECT Int
               | DELETE Int
```

- We now define the transformation function from Strings to the `Input` datatype.

```
strToInput : String -> Maybe Input
```

A way to transform string input into “commands” iii

An example application of this function is:
`strInput "query INSERT 1 A"` $\rightsquigarrow$ `QUERY INSERT 1 "A"`.

- Essentially, it should handle three classes of strings, one for each form of input. Note also that both `OPEN` and `QUERY` have *parameters*.
- Think about this function and try to figure it out by yourself!
- You may check my proposed solution later in the slides.
- I suggest two auxiliary functions: `mkQuery` and `parseQuery`.
  - Function `mkQuery : String -> Maybe (Input)` decomposes the input string and calls `parseQuery` to build the `Input` instance.

A way to transform string input into “commands” iv

- Function `parseQuery : List String -> Maybe Input` receives a list of strings, such as `["query", "INSERT 1 A"]`, and produces an instance of the `Input` datatype, such as `QUERY INSERT 1 "A"`.

A way to represent the database i

- We chose to represent the database as a map (`SortedMap`), available in the Idris distribution.

```
Database : Type
Database =  SortedMap Int String
```

- The type `Database` is imply a synonym to a map from integers to strings.
  - Of course we could relate richer structures with the map and even create a more realistic representation of a database.
  - This simple map should suffice given our pedagogical needs at this time.
- To use it we need to:

A way to represent the database ii

  - Import it in our program with: `import Data.SortedMap`
  - and invoke Idris using the command line: `idris -p contrib simple-app.lidr`

- This will inform the run-time that we are importing the `SortedMap` and where to find it (in package `contrib`).

A way to evaluate commands in the presence of a database i

- We define function by structural induction (the constructors `INSERT`, `SELECT` and `DELETE`) of the datatype `QueryLang` and relate each constructor with an operation of datatype `SortedMap`.
    - Of course, a more realistic implementation wouldn't define a simple bijection (one-to-one), but, again, enough for our pedagogical needs.
- This about it! You may *cheat* and look the proposal solution if you will. But think hard first!

A way to evaluate commands in the presence of a database ii

```
eval : QueryLang -> Database -> (Database, Report)
eval (INSERT i s) db = ?i
eval (SELECT i) db = ?s
eval (DELETE i) db = ?d
```

- Use the command `:browse Data.SortedMap` to learn about `SortedMap`'s interface.
- The notation `(Database, Report)` simply defines a *pair* of `Database` and `Report` where the latter is simply a list of strings.

An updated protocol datatype i

- As before, we have transitions to open, close and query the database.
- However, we now have a more refined notion of *state* of the database app (`DBState`) comprised by the name of open database, its connection status, the database itself and the report of the last query.
- We must update the type of the datatype and of its transitions.
- As always, think about it and cheat if you feel like it...
- A sorted map may be initialized with the `fromList` command. (Search for it in Idris' REPL.)

An updated protocol datatype ii

```
data DBCmd : Type -> DBState -> DBState -> Type
where
  OPENDB : (d : String) ->
   DBCmd () (s, NotConn, db, [])
            (..., ..., ..., ...)
  CLOSEDB :
   DBCmd () (s, Conn, db, r)
            (..., ..., ..., ...)
  QUERYDB : (q : QueryLang) ->
   DBCmd () (s, Conn, db, r)
     (s, Conn, fst (...), snd (...))
  Display : String -> DBCmd () st st
```

An updated protocol datatype iii

```
GetInput : DBCmd (Maybe Input) st st
Pure : ty -> DBCmd ty state state
(>>=) : DBCmd a state1 state2 ->
        (a -> DBCmd b state2 state3) ->
        DBCmd b state1 state3
```

An interactive user-interface i

- Streams are the way to go to write app with infinite data.
- This is exactly what happens when we write interactive applications.
- The datatype `DBIO` defines an stream of instances of `DBState`. Note the use of the `Inf` constructor, while defining a trace of `DBState` with the `Do` constructor...

```
data DBIO : DBState -> Type where
     Do : DBCmd a state1 state2 ->
          (a -> Inf (DBIO state2)) -> DBIO state1
```

- ... which is precisely what we need to implement the lifting of `(>>=)` to sequences of `DBCmd`.

An interactive user-interface ii

- Now we need to define a function that will interact with the user and enact the appropriate actions given a well-formed input. Function dbLoop does precisely that. Again, it is defined by cases on the possible states.
- Understand the following implementation and think about the missing cases captured by the ellipsis.

```
dbLoop : DBIO st
dbLoop {st = (n, NotConn, d, [])} =
 do Just x <- GetInput
            | Nothing =>
               do Display "Invalid input"
                  dbLoop
```

An interactive user-interface iii

```
        case x of
          ...
          otherwise =>
           do Display
                "You should open the database first."
              dbLoop

dbLoop {st = (n, Conn, d, r)} =
 do Just x <- GetInput
            | Nothing =>
               do Display "Invalid input"
                  dbLoop
    case x of
```

An interactive user-interface iv

```
CLOSE =>
 do CLOSEDB {s = n} {db = d}
    dbLoop
...
otherwise =>
 do Display
      "Either close or query the database."
    dbLoop
```

- Function dbLoop executes sequences of commands. We need to be able to "connect" it with the IO system of Idris' run time.
- From the user's perspective, dbLoop must be ran "forever". And that is precisely what main does.

An interactive user-interface v

```
main : IO ()
main =
 run forever
      (dbLoop
        {st = ("", NotConn, fromList [(0,"0")], [])})
```

- Function `run` makes the connection I mentioned above, relating `DBIO` instances with `IO` instances.

```
run : Fuel -> DBIO state -> IO ()
run (More fuel) (Do c f)
  = do res <- runMachine c
       run fuel (f res)
run Dry p = pure ()
```

An interactive user-interface vi

- Datatype `DBIO` is a sequence of DB commands. Function `run` only "iterates" over the infinite sequence of commands, processing it step-by-step by means of function `runMachine`.
- And it does it using the *lazy* datatype `Fuel` (that we studied before), that allows `run` to execute DB commands one step at the time, with a `DBIO` (infinite) sequence.
- Let us take a look at the `runMachine` function. It is defined by cases on `DBCmd` datatype. We will only study one of its cases. The remaining ones are for you think about.

An interactive user-interface vii

```
runMachine : DBCmd ty inState outState -> IO ty
runMachine
 {inState = (s, NotConn, db, [])}
 {outState = (s', Conn, (fromList [(0, "0")]), [])}
 (OPENDB s') =
 do
  putStrLn ("DB " ++ s' ++ " open")
  showDB (fromList [(0, "0")])
```

- Function `runMachine` relates a DB command and IO actions. In the case of command `OPENDB s`, where `s` is a string, denoting the name of the database, `runMachine` prints that the database, whose name was given, is open and lists the contents of an initialized database.

Simple app full listing

Datatypes i

```
> import Data.SortedMap
>
> namespace Database
>
>     data ConnState = NotConn | Conn
>
>     Report : Type
>     Report = List String
>
>     Database : Type
>     Database =  SortedMap Int String
```

Datatypes ii

```
>
>     DBState : Type
>     DBState = (String, ConnState, Database, Report)
>
>     data QueryLang = INSERT Int String
>                    | SELECT Int
>                    | DELETE Int
>
>     data Input = OPEN String
>                | CLOSE
>                | QUERY QueryLang
```

Function `mkInsert i`

```
>     mkInsert : List String -> Maybe QueryLang
>     mkInsert xs =
>      case tail' xs of
>       Just y =>
>        case y of
>          s1 :: [s2]  => Just (INSERT (cast s1) s2)
>          otherwise => Nothing
>       otherwise => Nothing
```

Function `mkSelect i`

```
>     mkSelect : List String -> Maybe QueryLang
>     mkSelect xs =
>      case tail' xs of
>       Just y =>
>        case y of
>         [s] => Just (SELECT (cast s))
>         otherwise => Nothing
>       otherwise => Nothing
```

Function `mkDelete i`

```
>     mkDelete : List String -> Maybe QueryLang
>     mkDelete xs =
>      case tail' xs of
>       Just y => case y of
>          [s] => Just (DELETE (cast s))
>          otherwise => Nothing
>       otherwise => Nothing
```

Function parseQuery i

```
>     parseQuery : List String -> Maybe Input
>     parseQuery xs =
>      case head' xs of
>       Just "INSERT" =>
>        case mkInsert(xs) of
>         Just q => Just (QUERY q)
>         Nothing => Nothing
>       Just "SELECT" =>
>        case mkSelect(xs) of
>         Just q => Just (QUERY q)
>         Nothing => Nothing
```

Function `parseQuery` ii

```
>         Just "DELETE" =>
>          case mkDelete(xs) of
>            Just q => Just (QUERY q)
>            Nothing => Nothing
>         otherwise => Nothing
```

Function mkQuery i

```
>      mkQuery : String -> Maybe (Input)
>      mkQuery "" = Nothing
>      mkQuery  s =
>       let h = head' (words s)
>       in
>        case h of
>         Just "query" =>
>          let xs = tail' (words s)
>          in case xs of
>               Just y => parseQuery(y)
```

Function `mkQuery` ii

```
>                otherwise => Nothing
>            otherwise => Nothing
```

Function strToInput i

```
>     strToInput : String -> Maybe Input
>     strToInput s =
>                 if ((head' (words s)) == (Just "open"))
>                 then
>                   let db = tail' (words s)
>                   in
>                       case db of
>                         Just d =>
>                           case d of
>                             [s'] => Just (OPEN s')
>                             otherwise => Nothing
```

Function strToInput ii

```
>                         otherwise => Nothing
>                 else
>                     if s == "close"
>                     then Just CLOSE
>                     else mkQuery(s)
```

Function `eval i`

```
>     eval : QueryLang -> Database -> (Database, Report)
>     eval (INSERT i s) db = ((insert i s db), [])
>     eval (SELECT i) db =
>         case lookup i db of
>             Just s => (db , [s])
>             otherwise => (db, [])
>     eval (DELETE i) db = ((delete i db), [])
```

Datatype DBCmd i

```
>     data DBCmd : Type -> DBState -> DBState -> Type
>     where
>       OPENDB : (d : String) ->
>        DBCmd () (s, NotConn, db, [])
>                 (d, Conn, (fromList [(0,"0")]), [])
>       CLOSEDB :
>        DBCmd () (s, Conn, db, r)
>                 ("", NotConn, db, [])
>       QUERYDB : (q : QueryLang) ->
>        DBCmd () (s, Conn, db, r)
>          (s, Conn, fst (eval q db), snd (eval q db))
```

Datatype DBCmd ii

```
>         Display : String -> DBCmd () st st
>         GetInput : DBCmd (Maybe Input) st st
>         Pure : ty -> DBCmd ty state state
>         (>>=) : DBCmd a state1 state2 ->
>                 (a -> DBCmd b state2 state3) ->
>                 DBCmd b state1 state3
```

Datatype DBIO i

```
>     data DBIO : DBState -> Type where
>          Do : DBCmd a state1 state2 ->
>               (a -> Inf (DBIO state2)) -> DBIO state1
```

Function `showDB i`

```
>       showDB : Database -> IO ()
>       showDB db =
>        if null db
>        then putStrLn ""
>        else
>         putStrLn (show (zip (keys db) (values db)))
```

Function `runMachine i`

```
>       runMachine : DBCmd ty inState outState -> IO ty
>       runMachine
>        {inState = (s, NotConn, db, [])}
>        {outState = (s', Conn, (fromList [(0, "0")]), [])}
>        (OPENDB s') =
>        do
>         putStrLn ("DB " ++ s' ++ " open")
>         showDB (fromList [(0, "0")])
```

Function `runMachine` ii

```
>      runMachine
>       {inState = (s, Conn, db, r)}
>       {outState = ("", NotConn, db, [])}
>       CLOSEDB = putStrLn ("DB " ++ s ++ " closed")
```

Function runMachine iii

```
>      runMachine
>       {inState = (s, Conn, db, r)}
>       {outState = (s, Conn,
>         (fst (eval q db)),
>         (snd (eval q db)))}
>       (QUERYDB q) =
>        do putStrLn("DB contents")
>           showDB   (fst (eval q db))
>           putStrLn("Query result")
>           putStrLn (unwords (snd (eval q db)))
```

Function `runMachine` iv

```
>     runMachine (Pure x) = pure x
>     runMachine (cmd >>= prog) = do x <- runMachine cmd
>                                    runMachine (prog x)
>     runMachine (Display str) = putStrLn str
>     runMachine {inState = (s, c, db, r)} GetInput
>       = do putStr ("DB: " ++ s ++ "> ")
>            x <- getLine
>            pure (strToInput x)
```

Fuel, forever and run i

```
>     data Fuel = Dry | More (Lazy Fuel)
>
>     partial
>     forever : Fuel
>     forever = More forever
>
>     run : Fuel -> DBIO state -> IO ()
>     run (More fuel) (Do c f)
>       = do res <- runMachine c
>            run fuel (f res)
>     run Dry p = pure ()
```

Function >>= lifted to streams of DBCmd i

```
>      namespace DBDo
>        (>>=) : DBCmd a state1 state2 ->
>                (a -> Inf (DBIO state2)) -> DBIO state1
>        (>>=) = Do
```

Function dbLoop i

```
>       dbLoop : DBIO st
>       dbLoop {st = (n, NotConn, d, [])} =
>        do Just x <- GetInput
>                   | Nothing =>
>                      do Display "Invalid input"
>                         dbLoop
>           case x of
>            OPEN x =>
>             do OPENDB x {db = d}
>                dbLoop
>            otherwise =>
```

Function dbLoop ii

```
>             do Display
>                  "You should open the database first."
>                dbLoop
>
>     dbLoop {st = (n, Conn, d, r)} =
>      do Just x <- GetInput
>                 | Nothing =>
>                    do Display "Invalid input"
>                       dbLoop
>         case x of
>          CLOSE =>
>           do CLOSEDB {s = n} {db = d}
>              dbLoop
```

Function dbLoop iii

```
>            (QUERY q) =>
>             do QUERYDB q {s = n} {db = d}
>                dbLoop
>            otherwise =>
>             do Display
>                 "Either close or query the database."
>                dbLoop
```

Function `main` i

```
> main : IO ()
> main =
>  run forever
>       (dbLoop {st = ("", NotConn,
>                      fromList [(0,"0")], [])})
```